PATENT
Docket No. **SYB/0101.01**

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Evan P. Ireland	Examiner: Wang, Rongfa Philip
Serial No.: 10/709,917	Art Unit: 2191
Filed: June 4, 2004	APPEAL BRIEF
For: Attribute-Based Component Programming System and Methodology for Object-Oriented Languages	

Mail Stop Appeal
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

BRIEF ON BEHALF OF EVAN P. IRELAND

This is an appeal from the Final Rejection mailed September 12, 2008, in which currently pending claims 1-61 stand finally rejected. Appellant filed a Notice of Appeal on December 12, 2008. This brief is submitted electronically in support of Appellant's appeal.

TABLE OF CONTENTS

1. REAL PARTY IN INTEREST

The real party in interest is assignee Sybase, Inc. located at One Sybase Drive, Dublin, CA 94568.

2. RELATED APPEALS AND INTERFERENCES

There are no appeals or interferences known to Appellant, the Appellant's legal representative, or assignee which will directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

3. STATUS OF CLAIMS

The status of all claims in the proceeding is as follows:

Rejected: 1-61

Allowed or Confirmed: None

Withdrawn: None

Objected to: None

Canceled: None

Identification of claims that are being appealed: Claims 1-61

An appendix setting forth the claims involved in the appeal is included as Section 8 of this brief.

4. STATUS OF AMENDMENTS

One Amendment and two Amendments After Final have been filed in this case. Appellant filed an Amendment on June 30, 2008 in response to a non-final Office Action dated March 31, 2008. In the Amendment filed on June 30, 2008, the pending claims were amended in a manner that Appellant believes clearly distinguished the claimed invention over the art of record, for overcoming the art rejections. In response to the Examiner's Final Rejection dated September 12, 2008 (hereinafter "Final Rejection") finally rejecting Appellant's claims, Appellant filed an Amendment After Final and Request for Reconsideration on December 9, 2008 in an effort to address the Examiner's non-prior art rejections of certain of Appellant's claims and to request reconsideration of

the art rejections. However, in an Advisory Action dated December 19, 2008, the Examiner indicated that Appellant's proposed amendments to the claims would not be entered. Appellant also filed an Amendment After Final on February 9, 2009 solely to address objections made by the Examiner to Appellant's specification. Thus, no amendments to the claims have been entered since the date of the Final Rejection.

5. SUMMARY OF CLAIMED SUBJECT MATTER

Appellant asserts that the art rejections herein fail to teach or suggest all of the claim limitations of Appellant's claimed invention, where the claimed invention is set forth in the embodiment in **independent claim 1:** A method for dynamically generating program code adding behavior to a program based on attributes (see generally, e.g., Appellant's specification paragraph [0021], paragraphs [0063]-[0065], paragraph [0072], paragraphs [0079]-[0085]; also see generally e.g., Fig. 5 at 501-506), the method comprising: adding a static field of type Component to a program class of the program to create a component (see e.g., Appellant's specification paragraph [0081], paragraphs [0095]-[0102]; Fig. 5 at 501 (Add static component field (object) to class from a component class of library); see also, paragraphs [0071]-[0075]), defining at least one attribute specifying declaratively behavior to be added to the program (see e.g., Appellant's specification paragraph [0021], paragraph [0083] (developer/user defines attributes for adding behavior to classes or methods of a program); Fig. 5 at 503; see also paragraphs [0118]-[0122]), wherein said at least one attribute comprises active metadata used to generate program code for inclusion in the program (see e.g., Appellant's specification, paragraph [0021], paragraph [0031] (attributes refers to language construct that programmers use to add additional information (i.e., metadata) to code elements), paragraph [0083], paragraph [0126] (attributes as active metadata); Fig. 5 at 503; see also paragraphs [0118]-[0124]), associating said at least one attribute with the component (see e.g., Appellant's specification paragraph [0021], paragraphs [0084]-[0085]; Fig. 5 at 504-505), and in response to instantiation of the component at runtime, generating a subclass based on the program class and said at least one attribute, the subclass including dynamically generated program code based on said at least one attribute (see e.g.,

Appellant's specification paragraph [0021], paragraphs [0085]-[0086]; Fig. 5 at 506; see also, paragraphs [0123]-[0124]).

Appellant additionally asserts that the art rejections herein fail to teach or suggest all of the claim limitations of Appellant claimed invention, where the claimed invention is set forth in the embodiment in **independent claim 26:** A system for dynamically generating program code based on declarative attributes (see e.g., Appellant's specification paragraph [0022], paragraphs [0063]-[0065]; see generally, Fig. 3, Fig. 4; also see generally paragraphs [0069]-[0075], the system comprising: a computer having a processor and memory (see e.g. Appellant's specification paragraph [0046], paragraph [0062]; Fig. 1, Fig. 2; see also, paragraphs [0056]-[0061]), a component module for creating a component from a program class based on adding a static field of type Component to the program class (see e.g., Appellant's specification paragraph [0022], paragraph [0069], paragraph [0071] (Component class), paragraph [0081], paragraphs [0095]-[0102]; Fig. 5 at 501 (Add static component field (object) to class from a component class of library), an attribute module for defining at least one declarative attribute specifying behavior to be added to the program class (see e.g., Appellant's specification paragraph [0022], paragraph [0069], paragraph [0072] (Attribute class for implementing attributes), paragraph [0083] (developer/user defines attributes for adding behavior to classes or methods of a program); Fig. 5 at 503; see also paragraphs [0118]-[0122]), and associating said at least one attribute with the component (see e.g., Appellant's specification paragraph [0022], paragraphs [0084]-[0085]; Fig. 5 at 504-505), wherein said at least one declarative attribute comprises active metadata used to generate program code (see e.g., Appellant's specification, paragraph [0022], paragraph [0031] (attributes refers to language construct that programmers use to add additional information (i.e., metadata) to code elements), paragraph [0083], paragraph [0126] (attributes as active metadata); Fig. 5 at 503; see also paragraphs [0118]-[0124]), and a module for generating a subclass of the program class in response to instantiation of the component, the subclass including dynamically generated program code based on said at least one declarative attribute (see e.g., Appellant's specification paragraph [0022], paragraphs [0085]-[0086]; Fig. 5 at 506; see also, paragraphs [0123]-[0124]).

Appellant further asserts that the art rejections herein fail to teach or suggest all of the claim limitations of Appellant claimed invention, where the claimed invention is set forth in the embodiment in **independent claim 47:** A method for adding behavior to an application without access to application source code (see generally, e.g., Appellant's specification paragraph [0023], paragraphs [0063]-[0065], paragraph [0072], paragraphs [0079]-[0085]; also see generally e.g., Fig. 5 at 501-506), the method comprising: defining at least one attribute specifying declaratively behavior which is desired to be added to an application without access to the application source code (see e.g., Appellant's specification paragraph [0023], paragraph [0083] (developer/user defines attributes for adding behavior to classes or methods of a program); Fig. 5 at 503; see also paragraphs [0118]-[0122]), wherein said at least one attribute comprises active metadata used to generate code adding behavior to the application (see e.g., Appellant's specification, paragraph [0023], paragraph [0031] (attributes refers to language construct that programmers use to add additional information (i.e., metadata) to code elements), paragraph [0083], paragraph [0126] (attributes as active metadata); Fig. 5 at 503; see also paragraphs [0118]-[0124]), storing said at least one attribute in a properties file external to the application (see e.g., Appellant's specification, paragraph [0023], paragraph [0078], paragraph [0083]; Fig. 4 at 420 (Properties File), creating a dynamic attributes class based on the properties file (see e.g., Appellant's specification, paragraph [0023], paragraph [0078], paragraph [0083], paragraph [0085]; Fig. 4 at 470 (Dynamic Attributes Class)), compiling the application and the dynamic attributes class (see e.g., Appellant's specification, paragraph [0023], paragraph [0078], paragraph [0083]; Fig. 4 at 430b (JAVAC – Java Compiler), 475, 450b (DJC - Dynamic Java Component Precompiler), 455b (RCC - Runtime Component Compiler)), and generating a subclass which includes dynamically generated code adding behavior to the application based on said at least one attribute (see e.g., Appellant's specification, paragraph [0023], paragraph [0078], paragraph [0083], paragraphs [0085]-[0086]; Fig. 4 at 480 (Component Subclass); see also, Fig. 5 at 506).

Appellant additionally argues based on **dependent claims 6 and 31** which include claim limitations pertaining to: defining at least one attribute based on comments

in source code of the program class (see e.g., Appellant's specification paragraph [0063], paragraph [0077] (attributes defined for class and included in comments in source code), paragraph [0083], paragraphs [0138]-[0139]; also see e.g., Fig. 4 at Fig. 4 at 410, 450, 440, 430a, 445, 450b, 455b, 480, 495).

6. GROUNDS OF REJECTION TO BE REVIEWED

The grounds for appeal are:

(1st) Whether claims 1-46 are unpatentable under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which Appellant regards as the invention.

(2nd) Whether claims 1-5, 15-19, 21-26, 28-30, 33, 40-43, and 45-46 are unpatentable under 35 U.S.C. Section 103(a) as being obvious U.S. Patent 6,182,277 to DeGroot et al (hereinafter "DeGroot") in view of U.S. Patent 5,913,063 to McGurrin et al (hereinafter "McGurrin").

(3rd) Whether claims 6-14, 20, 31-39, and 44 are unpatentable under 35 U.S.C. Section 103(a) as being obvious over DeGroot (above) in view of McGurrin (above) and further in view of U.S. Patent 7,103,885 of Foster (hereinafter "Foster").

(4th) Whether claims 47-61 are unpatentable under 35 U.S.C. 103(a) as being obvious over DeGroot (above) in view of McGurrin (above) and further in view of U.S. Published Application 2003/0055936 of Zhang et al (hereinafter "Zhang"). Appellant notes that although the Examiner has not specifically included McGurrin in the grounds of rejection at paragraph 11 on page 20 of the Final Rejection, the Examiner has referenced McGurrin in the text that follows (e.g., at page 21 of the Final Rejection) and, therefore, Appellant understands that the rejection of claims 47-61 is based on the combination of DeGroot, McGurrin and Zhang.

7. ARGUMENT

A. First Ground: Claims 1-46 rejected under 35 U.S.C. 112, second paragraph

1. General

Claims 1-46 stand rejected under 35 U.S.C. 112, second paragraph, as being

indefinite for failing to particularly point out and distinctly claim the subject matter which Appellant regards as the invention. As will be shown below, Appellant's claims 1-46 comply with the requirements of the second paragraph of 35 U.S.C. Section 112 as they point out and distinctly claim the subject matter of Appellant's invention.

2. Claims 1-46

The Examiner rejected claims 1-46 on the basis that independent claims 1 and 27 include claim limitations of a "static field of type Component" without any description of what "type Component" is (Final Rejection paragraph 8, page 3). Appellant respectfully believes the meaning of the above claim limitation is clear when read in context with the balance of Appellant's claim. For instance, Appellant's claim 1 includes the following claim limitations:

> A method for dynamically generating program code adding behavior to a program based on attributes, the method comprising:
> adding a static field of type Component to a program class of the program to create a component;
> defining at least one attribute specifying declaratively behavior to be added to the program, wherein said at least one attribute comprises active metadata used to generate program code for inclusion in the program;
> associating said at least one attribute with the component; and
> in response to instantiation of the component at runtime, generating a subclass based on the program class and said at least one attribute, the subclass including dynamically generated program code based on said at least one attribute.
>
> (Appellant's claim 1, emphasis added)

As discussed below in more detail, in the prior art rejections of Appellant's claimed invention the Examiner equates DeGroot's declarative programming techniques for object-oriented environments to Appellant's attribute-based component programming system and methodology. DeGroot's declarative programming technique permits augmenting the functionality of a method on an object with "rules" (see e.g., DeGroot, Abstract). DeGroot's "rules" are high-level requirements or functions that augment the functionality of methods of an object (see e.g., DeGroot, column 4, lines 40-43). In response to the first Office Action in this case and in order to further clarify Appellant's invention, Appellant modified claims 1 and 27 to more specifically define that the

process of creating a component includes adding a "static field" called "component" whose data type is "Component" (e.g., public static final Component = new Component(MyExample.class). Thus, the addition of claim limitations including a "static field" of type "Component" to a class provides more specific claim limitations that cannot be construed as in any way constituting a "rule" that augments the functionality of methods of an object as described by DeGroot. These features are also specifically described in Appellant's specification including, for instance, at paragraph [0084] as follows:

> At step 501, a developer/user may provide for creating a component from the class he or she is developing (e.g., a plain Java class named "MyClass") by the addition to the class of a static component field (or object) from a "Component" class.
>
> (Appellant's specification, paragraph [0084])

This is also described, for example, in paragraphs [0095]-[0096] of Appellant's specification which provides that making a class into a component is accomplished by the addition to the class of a static field of type Component as follows: public static final Component component = new Component(MyClass.class). Creation of the component in this fashion facilitates dynamic generation of a subclass when the component is instantiated (see e.g., the example illustrated in Appellant's specification at paragraphs [0098]-[0112] and another example illustrated at paragraphs [0119]-[0125] of Appellant's specification).

Additionally, although Appellant believes the meaning of the claims 1 and 27 (and dependent claims thereof) to be clear, in Appellant's Amendment after Final, Appellant requested the Examiner to enter amendments to claims 1 and 27 deleting the words "of type Component" so as to address the Examiner's rejection. However, the Examiner refused to enter this amendment, purportedly on the basis that it would represent a change in scope of the claims and therefore raise new issues that would require further consideration and/or search (Examiner Advisory Action dated December 19, 2008, note 3). However, the words "of type Component" were added in Appellant's

Amendment filed on June 30, 2008 and were not included in the original version of Appellant's claims 1 and 27. Thus, Appellant respectfully believes the Examiner's rationale for refusing to enter the amendments requested by Appellant is incorrect and inappropriate given that original versions of these claims to which the Examiner responded in the first Office Action did not include these limitations. Furthermore, Appellant believes that the Examiner's refusal to enter the minor amendments to these claims is contrary to 37 CFR Section 1.116 (b) (1) which provides that an amendment may be made canceling claims or complying with any requirement of form expressly set forth in a previous Office action after a final rejection or other final action. Additionally, the effect of the amendments requested by Appellant would be remove issues for Appeal which is encouraged under the relevant rules (see e.g., MPEP Section 714.13).

Thus, for the reasons set forth above, Appellant respectfully believes that the rejection of claims 1-46 under Section 112, second paragraph should not be sustained.

B. Second Ground: Claims 1-5, 15-19, 21-26, 28-30, 33, 40-43, and 45-46 rejected under 35 U.S.C. 103(a)

1. General

Under Section 103(a), a patent may not be obtained if the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art to which the subject matter pertains. To establish a prima facie case of obviousness under this section, the Examiner must establish: (1) that there is some suggestion or motivation, either in the references themselves or in the knowledge generally available to one of ordinary skill in the art, to modify the reference or to combine reference teachings, (2) that there is a reasonable expectation of success, and (3) that the prior art reference (or references when combined) must teach or suggest all the claim limitations. (See e.g., MPEP 2142). The reference(s) cited by the Examiner fail to meet these conditions.

2. Claims 1-5, 15-19, 21-26, 28-30, 33, 40-43, and 45-46

Claims 1-5, 15-19, 21-26, 28-30, 33, 40-43, and 45-46 stand rejected under 35 U.S.C. 103(a) as being unpatentable over U.S. Patent 6,182,277 to DeGroot et al

(hereinafter "DeGroot") in view of U.S. Patent 5,913,063 to McGurrin et al (hereinafter "McGurrin").

By way of background, Appellant's invention provides for use of "attributes" as a flexible and extensible way of modifying the behavior of a program (or a class or method of a program). As described below in more detail, the "attributes" of Appellant's invention are a language construct that developers (users) can use to add metadata to code elements (e.g., assemblies, modules, members, types, return values, and parameters) to extend their functionality. A developer (user) can utilize Appellant's attribute-based component programming system to define attributes that specify declaratively the addition of certain behavior to a program being developed (see e.g., Appellant's specification, paragraph [0031]). These attributes comprise "active" metadata used to generate code for adding behavior to a program as it executes at runtime. For example, a developer can use Appellant's solution to define attributes that add tracing behavior to a program module under development as hereinafter described in more detail. The Examiner contends that DeGroot and McGurrin include comparable teachings; however, review of these references finds that they are distinguishable in a number of respects as hereinafter described.

The Examiner equates Appellant's attribute-based component programming system and methodology to DeGroot's declarative programming techniques for object oriented environments. DeGroot's declarative programming technique permits augmenting the functionality of a method on an object with "rules" (see e.g., DeGroot, Abstract). With DeGroot's system a user submits declarative statements that are associated with a method identified on an object so that when the method on the object is called the declarative statements are executed in addition to the methods on the object (see e.g., DeGroot, Abstract). Accordingly, DeGroot's "rules" are high level requirements, parameters or constraints that set forth requirements or parameters that define object behavior (see e.g., DeGroot, column 4, lines 40-41). DeGroot's rules may, for example, define the range of parameters acceptable for execution of a method on an object. For instance, a rule for a "time" method may check whether the value stored in the "day" attribute is a valid day for the corresponding "month" parameter (e.g., the day

parameter is a value between 1 and 30 for the month of September) (DeGroot, col. 5, lines 32-40). As another example involving an order entry application program, a user of DeGroot's system may specify the requirement that before the "ship" method is executed, the shipping information, used by the ship method, must exist (DeGroot, col. 5, lines 41-48). As illustrated by these examples, the **rules** described by DeGroot comprise constraints or requirements associated with a given method of an object and, therefore, are fundamentally different from the **attributes** of Appellant's invention.

Appellant's attribute-based component programming system and methodology provides a flexible and extensible way of modifying the behavior of a program or a class or a method of a program. Appellant's attributes are used to generate additional code for inclusion in a program class or subclass. For instance, a developer may define attributes that specify the addition of method tracing behavior to a class being developed in order to assist in debugging the program that is being developed (see e.g., Appellant's specification, paragraph [0080]). This tracing behavior may, for instance, cause a message to be displayed on the screen indicating that a particular method was called, specifying the calling parameters with which the method was called, and providing the result returned by the method (see e.g., Appellant's specification, paragraph [0083]). Attributes that are defined and added to a class or method result in additional code being automatically generated to implement the additional behavior (e.g., method tracing behavior) when the method is called at runtime (see e.g., Appellant's specification, paragraph [0084]). For example, a developer may add the above-described tracing attributes to an "add" method of a given class, so that when the "add" method is called, the defined attribute causes the tracing behavior to be added. Thus, the attributes of Appellant's invention are substantially different from DeGroot's rules that define requirements or constraints. The distinctive features are also included as limitations of Appellant's claims. For instance, Appellant's claim 1 includes the following claim limitations:

> A method for dynamically generating program code adding behavior to a program based on attributes, the method comprising:
> adding a static field of type Component to a program class of the program to

create a component;
defining at least one attribute specifying declaratively behavior to be added to the program, wherein said at least one attribute comprises active metadata used to generate program code for inclusion in the program;
associating said at least one attribute with the component; and
in response to instantiation of the component at runtime, generating a subclass based on the program class and said at least one attribute, the subclass including dynamically generated program code based on said at least one attribute

(Appellant's claim 1, emphasis added)

More particularly, Appellant's claimed invention specifically provides for associating the defined attributes with a component so as to dynamically generate a **subclass** containing the additional code that adds the desired behavior to the program. This is described, for example, as follows:

> The attributes added to the class typically result in the generation of extra code that is added into the dynamically generated subclasses. The Component class generates the subclass (i.e., creates the shell) for each of the components (i.e., component objects) that are instantiated. For each of the component objects in the class, the Component class essentially asks the attributes to specify the extra code that is to be inserted into the appropriate methods of the class.
>
> (Appellant's specification, paragraph [0085], emphasis added).

In contrast, DeGroot's solution does not dynamically generate additional code to extend program functionality. Instead, declarative statements written by the developer are called in addition to the methods on the object (see e.g., DeGroot, col. 4, lines 1-5). Additionally, in DeGroot's system associating declarative statements (or rules) with a method involves redirecting a call to the method to a "pinch-point" operation (see e.g., DeGroot col. 11, lines 55-63). The "pinch-point" operation determines whether there are any rules associated with the method called (DeGroot, col. 5, lines 14-21). If rules are determined to exist, then one or declarative statements associated with the method are called (DeGroot, col. 5, lines 21-29; see also, claim 8).

In contrast, Appellant has no equivalent notion of "pinch-point"; nothing is dynamically determined when a method having associated attributes is called. So although at a high level DeGroot describes a "declarative" system, it could more

accurately be described as being based on "dynamic rule evaluation leading to declarative statement execution". This differs dramatically from Appellant's invention which generates subclass code based on active metadata (i.e., attributes).

This brings up another difference between Appellant's invention and DeGroot's solution. In response to instantiation of a component at runtime, Appellant's invention generates a subclass based on the program class and the defined attribute. The subclass includes dynamically generated program code based on the defined attribute. In contrast, DeGroot teaches away from this aspect of Appellant's invention as DeGroot rejects the notion of subclassing. DeGroot notes that subclassing technique requires re-compiling the code, which in turn requires shutting down the application and reloading both the new and old code. Thus, DeGroot looks for another solution, stating "it is desirable to provide a technique to permit augmentation of a method, to alter the behavior of an object, even though the source metadata is not available...." (DeGroot, col. 2, lines 43-46). DeGroot proceeds to then describe a system that is implemented without use of subclassing.

In the Final Rejection the Examiner acknowledges that DeGroot does not include teachings of generating a subclass based on a program class and at least one attribute, wherein the subclass includes dynamically generated program code based on such attribute(s) (Final Rejection, pages 4-5). Therefore, the Examiner adds McGurrin as providing these teachings.

However, turning to McGurrin one finds that its teachings are not at all analogous to Appellant's claimed invention and offer nothing that would overcome DeGroot's rejection of the notion of sub-classing nor anything that in combination would teach or suggest Appellant's claimed invention. Further, although McGurrin does refer to an "attribute", the attribute described by McGurrin is something completely different from the attributes of Appellant's invention. McGurrin's attribute is also commonly known (e.g. in the Java programming language) as a "field" and is clearly described in this manner by McGurrin such as, for example, as follows:

> In object oriented programming, the world is modeled in terms of objects. An object is a record combined with the procedures and functions that manipulate it. All objects in an object class have the same fields ("attributes"), and are

> manipulated by the same procedures and functions ("methods"). An object is said to be an "instance" of the object class to which it belongs.
> Sometimes an application requires the use of object classes that are similar, but not identical. For example, the object classes used to model both dolphins and dogs might include the attributes of nose, mouth, length and age. However, the dog object class may require a hair color attribute, while the dolphin object class requires a fin size attribute.

(McGurrin, col. 1, lines 10-25, emphasis added).

McGurrin goes on to describe that an object class may inherit attributes from another class. For example, both the "dog" and "dolphin" classes may inherit from an "animal" class so as to avoid having to duplicate code for the same attributes (e.g., nose, mouth, length, age) in multiple classes. McGurrin also describes the well-known technique of visually establishing relationships between attributes of objects in a visual programming environment. In other development environments, these attributes may be known as a "control" (if visual) or "instance variable" (if non-visual) rather than a field; however all of these are distinguishable from the attributes of Appellant's invention.

The attributes of Appellant's invention are defined as something completely different from those described by McGurrin. Appellant's "attributes" are a language construct that programmers (developers) use to add additional information (i.e., metadata) to code elements (e.g., assemblies, modules, members, types, return values, and parameters) to extend their functionality (see e.g., Appellant's specification, paragraph [0031]). These attributes are used by Appellant's invention to provide a flexible and extensible way of modifying the behavior of a program or a class or a method of a program (see e.g., Appellant's specification, paragraph [0031]). A developer can utilize Appellant's attribute-based component programming system to define attributes that specify declaratively the addition of certain behavior to a program being developed. These attributes comprise "active" metadata used to generate code for adding behavior to a program as it executes at runtime. These features of attributes comprising "active" metadata for generating other code for inclusion in a program subclass are also included as limitations of Appellant's claims. For example, Appellant's claim 1 includes the following claim limitations:

defining at least one attribute specifying declaratively behavior to be added to the program, wherein said at least one attribute comprises active metadata used to generate program code for inclusion in the program;
associating said at least one attribute with the component; and
in response to instantiation of the component at runtime, generating a subclass based on the program class and said at least one attribute, the subclass including dynamically generated program code based on said at least one attribute.

(Appellant's claim 1, emphasis added)

With Appellant's claimed invention, attributes that are defined and added to a class or method result in additional code being automatically generated to implement the additional behavior (e.g., method tracing behavior) when the method is called at runtime (see e.g., Appellant's specification, paragraph [0084]). For instance, a developer may define attributes that specify the addition of method tracing behavior to a class being developed in order to assist in debugging the program that is being developed (see e.g., Appellant's specification, paragraph [0080], paragraph [0083]). The developer may, for example, add the tracing attributes to an "add" method of a given class, so that when the "add" method is called, the defined attribute causes the tracing behavior to be added to the class. Thus, the "attributes" of Appellant's invention that comprise active metadata for generating code for inclusion in a program are fundamentally different both from the "fields" of McGurrin as well as from DeGroot's "rules" that define requirements or constraints.

The attributes or Appellant's invention are used to generate a subclass based on the original class which includes extra code that is added into the dynamically generated subclasses based on the attributes as described above. DeGroot does not include any comparable teachings of generating additional code to extend program functionality based on attributes. Although McGurrin describes subclassing in general terms, McGurrin also fails to teach or suggest dynamically generating program code based on defined attributes. Instead, McGurrin simply describes visual coding tools for generating code in response to user input (e.g., a user drawing a particular element on screen) as follows:

> With a visual coding tool, some or all of the source code of a software application is generated automatically based on user manipulation of visual elements displayed on a computer screen….
> Some visual coding tools generate source code for object oriented programming languages. With these visual coding tools, a generic object class is typically defined for every type of visual element. When a user draws a particular visual element, the visual coding tool generates source code for (1) a new object class that inherits from the corresponding generic object class, where the attributes of the subclass are initially set to those reflected in the element drawn by a user, and (2) an instance of the new object class.
> For example, assume that a user draws a window that has a particular size, color and location within a visual coding environment. The visual coding tool will typically generate (1) source code that defines a subclass of a generic window class and sets the initial values of the size, color and location attributes of the subclass to values that correspond to the size color and location of window drawn by the user, and (2) source code that declares an instance of the new window subclass.

(McGurrin, col. 2, lines 14-45, emphasis added)

As shown above, with McGurrin's solution code is generated in response to user interaction with a visual user interface of the program. For example, in response to a user drawing a particular element, code is generated in a subclass and attributes of the subclass and sets the initial size, color and location attributes of the subclass to the values corresponding to those drawn by the user. Thus, McGurrin is not adding code through the attributes but rather is providing data values for attributes that already exist as a part of a subclass. In contrast, Appellant's attributes specify declaratively behavior to be added to the program.

3. Conclusion

All told, the combined references provide no teaching or suggestion of defining attributes comprising active metadata and using these attributes for dynamically generating subclasses including code for adding behavior to a program as it executes at runtime. Therefore, as DeGroot and McGurrin, even when combined, do not teach or suggest all of the claim limitations of Appellant's claims 1-5, 15-19, 21-26, 28-30, 33, 40-43, and 45-46 (and other claims) it is respectfully submitted that the claims distinguish over the combined references and the rejection under Section 103 should not be

sustained.

C. Third Ground: Claims 6-14, 20, 31-39, and 44 rejected under 35 U.S.C. 103(a)

1. Claims 6-14, 20, 31-39, and 44

Claims 6-14, 20, 31-39, and 44 stand rejected under 35 U.S.C. 103(a) as being unpatentable over DeGroot (above) in view of McGurrin (above) and further in view of U.S. Patent 7,103,885 of Foster (hereinafter "Foster").

Appellant's claims are believed to be allowable for at least the reasons cited above in Appellant's **Second Ground** of Appeal (hereby incorporated by reference) pertaining to the deficiencies of DeGroot and McGurrin as to Appellant's claimed invention. As these claims are dependent upon, and incorporate the limitations of Appellant's independent claims, they are distinguishable for the reasons previously described in detail. As Foster does not provide any teaching of defining attributes comprising active metadata used to generate code for adding behavior to a program as it executes at runtime, it does not cure the deficiencies of the DeGroot and McGurrin references as to Appellant's invention.

Foster's solution comprises a source code version management system. Although Foster does discuss "attributes" associated with source code files, the attributes described by Foster are represented by the presence (or not) of an attribute tag and an associated value in the comment field of a version management file (Foster col. 4, lines 10-13). For example, the software module may be a data file and one attribute may be whether or not the data file may be translated (Foster col. 4, lines 17-19). The software module is then processed based on whether a predetermined tag is found in the comment field of the version management file and, if it is found, the value associated with the tag (Foster col. 4, lines 23-26). For example, if no tag is found then the file is not translated (Foster col. 5, lines 1-5). Clearly, the attributes described by Foster (as well as the solution environment in which they are utilized) are dramatically different than the attributes of Appellant's claimed invention discussed in detail above. Furthermore, Appellant's review of Foster finds no teaching of using the attributes as the basis for generating code for adding behavior to a software program as it executes at runtime.

Additionally, the Examiner references Foster for the specific teachings of defining attributes based on comments in the source code of a program class (Final Rejection paragraph 10, page 14). By way of background, with Appellant's invention, a user may include attributes as comments in the source code (see e.g., Appellant's specification paragraph [0077]; see also, Fig. 4 at 410). An attributes class is generated based on these comments and when compiled with the base class, a subclass is generated which includes not only the functions provided in the base class, but also additional behavior based on the attributes defined for the class and included in the source code comments (see e.g., Appellant's specification paragraph [0077]; see also, Fig. 4 at 410, 450, 440, 430a, 445, 450b, 455b, 480, 495).

Even assuming that Foster's attributes are somehow comparable to those of Appellant's invention (which Appellant does not agree is correct), review of Foster finds its teachings about including attributes as comments in the source code are not comparable to those of Appellant's invention. Foster describes a solution for processing software modules in which each software module has a version management file associated with it (Foster Abstract and column 7, lines 51-58). With Foster's system, attributes are not included as comments in the source code of the program, but rather are included as tags in a version management file. (Foster column 7, lines 53-58). Thus, these teachings are not comparable to Appellant's claim limitations of attributes included as comments in the source code of the program itself (see e.g., Appellant's claim 6).

2. Conclusion

All told, the combined references do not teach or suggest defining attributes which comprise active metadata and using such attributes for generating code for extending the functionality of a software program. Additionally, the references provide no teaching or suggestion of attributes defined in source code comments of the program itself. Accordingly, as the combined references do not teach or suggest all of the limitation of Appellant's claims, it is respectfully submitted that claims 6-14, 20, 31-39, and 44 distinguish over the combined references and the rejection under Section 103 should not be sustained.

D. Fourth Ground: Claims 47-61 rejected under 35 U.S.C. 103(a)

1. Claims 47-61

Claims 47-61 stand rejected under 35 U.S.C. 103(a) as being unpatentable over DeGroot (above) in view of McGurrin (above) and further in view of U.S. Published Application 2003/0055936 of Zhang et al (hereinafter "Zhang"). As previously mentioned, although the Examiner has not specifically included McGurrin in the grounds of rejection at paragraph 11 on page 20 of the Final Rejection, the Examiner has referenced McGurrin in the text that follows (e.g., at page 21 of the Final Rejection) and, therefore, Appellant understands that the rejection of claims 47-61 is based on DeGroot, McGurrin and Zhang.

Appellant's claims are believed to be allowable for at least the reasons cited above in Appellant's **Second Ground** of Appeal (hereby incorporated by reference) pertaining to the deficiencies of the DeGroot and McGurrin references as to Appellant's claimed invention. Zhang does not cure these deficiencies as it includes no teaching or suggestion of defining attributes which comprise active metadata and using such attributes for dynamically generating subclasses including code that extends the functionality of a software program. Zhang is also distinguishable from Appellant's claimed invention for the additional reasons discussed below.

The Examiner acknowledges that DeGroot and McGurrin do not disclose the following limitations of Appellant's claim 47:

> storing said at least one attribute in a properties file external to the application;
> creating a dynamic attributes class based on the properties file;
> compiling the application and the dynamic attributes class
>
> (Appellant's claim 47)

To address these deficiencies, the Examiner adds Zhang as providing the relevant teachings. However, Zhang's solution relates generally to network software testing and, more particularly, provides for customizing attributes of a distributed processing system in a distributed test framework (see e.g., Zhang, Abstract, paragraph [0004]). The "dynamic attributes" described by Zhang are used to describe characteristics of a

processing resource (see e.g., Zhang, paragraph [0010]). For instance, the hardware and software configuration of a given machine to be used for testing are designated using Jini attributes (see e.g., Zhang, paragraph [0064]).

Particularly, the embodiments of Zhang's solution register the hardware and software configuration for each test system with a Jini look up service using Jini attributes. (By way of background, Jini is a network architecture for the construction of distributed systems in the form of modular co-operating services (see e.g., Zhang, paragraphs [0025]-[0028])). For example, some tests systems may run the Sun Solaris operating system while others may run Wintel (Windows) or Linux (see e.g., Zhang paragraph [0055]). The attributes enable the test system to locate a suitable test system from running particular tests such as, for instance, running a Linux test suite on a Linux machine and a Wintel test suite on a Wintel machine (see e.g., Zhang paragraph [0054]; see also paragraph [0060]). Thus, the "attributes" of Zhang's invention allow users to define characteristics of a test system or test being performed by a test service (see e.g., Zhang paragraph [0065]). For example, a user may define an attribute indicating that the Linux operating system is running on a particular test system as "Software.OS.Name=Linux." Thus, it is clear from this description that the attributes described by Zhang are distinguishable from the attributes of Appellant's invention as Zhang's attributes do not comprise active metadata for adding behavior to a program. Additionally, Zhang's system uses its attributes in a different fashion. For example, the above-mentioned attribute "Software.OS.Name" is provided to a lookup service of Zhang's test system which makes the attribute available for view by the system controller (Zhang, paragraph [0066]).

Zhang's solution also allows users to (optionally) define "dynamic attribute classes" such as one defining the type of Linux executing on a target system (Zhang, paragraph [0072]). To make this new dynamic attribute class available to the agent process of the test system, the user adds the new dynamic attribute class name to a "dynamic attribute list" (see e.g., Zhang paragraph [0078]). This attribute list is then read by the agent process of the test system to determine the location of, and load, the new dynamic attribute class which is then added to the attributes of the test system (see e.g.,

Zhang paragraph [0078]). Thus, while Zhang's "attribute list" lists certain dynamic attribute classes that a user may write to define characteristics of a given test system, Zhang makes no mention of compiling the application and the dynamic attributes class so as to generate a subclass which includes dynamically generated code adding behavior to the application as provided, for instance as claim limitations of Appellant's claim 47. A particular advantage of Appellant's approach is that access to the underlying source code of the application class is not required in order to add behavior to the program. A user may define attributes external to the class (e.g., in a properties file) and use these attributes to add behavior to the class (see e.g., Appellant's specification, paragraph [0078]). This feature is particularly useful in a case in which an application program is in a production environment and the source code is unavailable. Zhang's solution includes no comparable features.

2. Conclusion

The combined references provide no teaching or suggestion of defining attributes comprising active metadata and using these attributes for dynamically generating subclasses including code for adding behavior to a program as it executes at runtime. Although Zhang describes an attribute list that lists certain attribute classes, Zhang's attributes simply describe characteristics of a particular system (e.g., the operating system it runs) that are not at all comparable to the attributes of Appellant's claimed invention. Moreover, Zhang's attributes are not used to dynamically generate code which adds behavior to an application, but instead are used to indicate characteristics of a given hardware and software test system to a test program so that appropriate tests can be run on the test system. Therefore, as the combined references do not teach or suggest all of the claim limitations of Appellant's claims 47-61 (and other claims) it is respectfully submitted that the claims distinguish over the combined references and the rejection under Section 103 should not be sustained.

E. Conclusion

Appellant's invention greatly improves the efficiency of the task of adding behavior to an application even in cases where the application source code may not be

available. It is respectfully submitted that the present invention, as set forth in the pending claims, sets forth a patentable advance over the art.

In view of the above, it is respectfully submitted that the Examiner's rejection of Appellant's claims 1-46 under Section 112, second paragraph and claims 1-61 under Section 103 should not be sustained. If needed, Appellant's undersigned attorney can be reached at 925 465 0361. For the fee due for this Appeal Brief, please refer to the attached Fee Transmittal Sheet. This Appeal Brief is submitted electronically in support of Appellant's Appeal.

Respectfully submitted,

Date: February 10, 2009

/G. Mack Riddle/

G. Mack Riddle; Reg. No. 55,572
Attorney of Record

925 465 0361
925-465-8143 FAX

8. CLAIMS APPENDIX

1. A method for dynamically generating program code adding behavior to a program based on attributes, the method comprising:

adding a static field of type Component to a program class of the program to create a component;

defining at least one attribute specifying declaratively behavior to be added to the program, wherein said at least one attribute comprises active metadata used to generate program code for inclusion in the program;

associating said at least one attribute with the component; and

in response to instantiation of the component at runtime, generating a subclass based on the program class and said at least one attribute, the subclass including dynamically generated program code based on said at least one attribute.

2. The method of claim 1, wherein said defining step includes defining a particular attribute using active metadata, so as to provide a mechanism for generation of program code from said particular attribute.

3. The method of claim 2, wherein said active metadata dynamically generates code for inclusion in a subclass based on the program class.

4. The method of claim 1, wherein the generating step includes generating a subclass comprising an instance of a declared component class.

5. The method of claim 1, wherein the generating step includes generating a subclass comprising an instance of component class declared as abstract.

6. The method of claim 1, wherein said defining step includes defining at least one attribute based on comments in source code of the program class.

7. The method of claim 6, further comprising:

precompiling a class containing static attributes from said comments.

8. The method of claim 7, further comprising:

loading the class containing static attributes before subclass generation when a component is instantiated.

9. The method of claim 6, further comprising:

defining an automated mapping between attribute syntax in comments and attribute syntax as expressed in generated program code.

10. The method of claim 1, wherein said defining step includes defining at least one attribute in a property file external to the program class.

11. The method of claim 10, further comprising:

compiling a class containing dynamic attributes from said property file.

12. The method of claim 11, further comprising:

loading the class containing dynamic attributes before subclass generation when a component is instantiated.

13. The method of claim 10, further comprising:

defining an automated mapping between attribute syntax in a property file and attribute syntax as expressed in generated program code.

14. The method of claim 10, wherein attributes in the property file comprise property name and property value pairs.

15. The method of claim 1, wherein said defining step includes defining attributes for a superclass from which the program class inherits.

16. The method of claim 1, wherein said defining step includes defining attributes for the program class' package from which the program class inherits.

17. The method of claim 1, wherein said defining step includes defining attributes for an interface from which the program class inherits.

18. The method of claim 17, wherein said generating step includes generating an instance of a subclass to mock the behavior of the interface.

19. The method of claim 1, wherein said generating step includes generating code for a non-abstract method body based on an attribute defined for an abstract method.

20. The method of claim 1, wherein said generating step includes generating program code based on comments in a source file.

21. The method of claim 1, further comprising:

adding expected calls as instances of anonymous inner classes of the program; and

applying runtime introspection by a generated subclass to verify a sequence of expected calls.

22. The method of claim 1, wherein the component registers itself with a repository when the component is initially activated.

23. The method of claim 22, wherein the repository can be queried to determine components that are active.

24. The method of claim 1, further comprising:

a computer-readable medium having processor-executable instructions for

performing the method of claim 1.

25. The method of claim 1, further comprising:

a downloadable set of processor-executable instructions for performing the method of claim 1.

26. A system for dynamically generating program code based on declarative attributes, the system comprising:

a computer having a processor and memory;

a component module for creating a component from a program class based on adding a static field of type Component to the program class;

an attribute module for defining at least one declarative attribute specifying behavior to be added to the program class and associating said at least one attribute with the component, wherein said at least one declarative attribute comprises active metadata used to generate program code; and

a module for generating a subclass of the program class in response to instantiation of the component, the subclass including dynamically generated program code based on said at least one declarative attribute.

27. The system of claim 26, wherein the subclass adds tracing behavior to a program.

28. The system of claim 26, wherein the subclass is a subclass of an abstract class.

29. The system of claim 26, wherein said at least one declarative attribute includes active metadata, so as to provide a mechanism for generation of program code.

30. The system of claim 29, wherein said active metadata dynamically generates code for inclusion in the subclass of the program class.

31. The system of claim 26, wherein the attribute module provides for defining at least one attribute based on comments in source code of the program class.

32. The system of claim 31, further comprising:

a precomplier for precompiling a class containing static attributes from said comments.

33. The system of claim 32, wherein the module for generating loads the class containing static attributes before subclass generation.

34. The system of claim 31, further comprising:

an automated mapping between attribute syntax in comments and attribute syntax as expressed in generated program code.

35. The system of claim 26, wherein the attributed module provides for defining at least one attribute in a property file external to the program class.

36. The system of claim 35, further comprising:

a module for compiling a class containing dynamic attributes from said property file.

37. The system of claim 36, wherein the module for generating loads the class containing dynamic attributes before subclass generation when a component is instantiated.

38. The system of claim 35, further comprising:

an automated mapping between attribute syntax in a property file and attribute syntax as expressed in generated program code.

39. The system of claim 35, wherein attributes in a property file comprise property name and property value pairs.

40. The system of claim 26, wherein the attribute module provides for defining attributes for a superclass from which the program class inherits.

41. The system of claim 26, wherein the attribute module for provides for defining attributes for an interface from which the program class inherits.

42. The system of claim 41, wherein the module for generating generates an instance of a subclass to mock the behavior of the interface.

43. The system of claim 26, wherein the module for generating generates code for a non-abstract system body based on an attribute defined for an abstract method.

44. The system of claim 26, wherein the module for generating includes generating program code based on comments in a source file.

45. The system of claim 26, wherein the component registers itself with a repository when the component is initially activated.

46. The system of claim 45, wherein the repository can be queried to determine components that are active.

47. A method for adding behavior to an application without access to application source code, the method comprising:

defining at least one attribute specifying declaratively behavior which is desired to be added to an application without access to the application source code, wherein said at least one attribute comprises active metadata used to generate code adding behavior to the application;

storing said at least one attribute in a properties file external to the application;

creating a dynamic attributes class based on the properties file;

compiling the application and the dynamic attributes class; and

generating a subclass which includes dynamically generated code adding behavior to the application based on said at least one attribute.

48. The method of claim 47, wherein said defining step includes defining a particular attribute using active metadata, so as to provide a mechanism for generation of program code from said particular attribute.

49. The method of claim 48, wherein said active metadata dynamically generates code for inclusion in the subclass.

50. The method of claim 47, wherein said defining step includes defining an attribute for overriding a method of the application.

51. The method of claim 47, wherein said defining step includes defining an attribute for extending a method of the application.

52. The method of claim 47, further comprising:

loading the class containing dynamic attributes before generating the subclass.

53. The method of claim 47, further comprising:

defining an automated mapping between attribute syntax in the properties file and attribute syntax as expressed in generated program code.

54. The method of claim 47, wherein said at least one attribute in the properties file comprise property name and property value pairs.

55. The method of claim 47, wherein said creating step includes creating a

dynamic attributes class using a precompiler.

56. The method of claim 47, wherein said creating step includes creating a dynamic attributes class at runtime.

57. The method of claim 47, wherein said compiling step includes using a Java compiler (JAVAC).

58. The method of claim 47, wherein said generating step includes using a precompiler.

59. The method of claim 47, wherein said generating step includes using a runtime compiler.

60. The method of claim 47, further comprising:

a computer-readable medium having processor-executable instructions for performing the method of claim 47.

61. The method of claim 47, further comprising:

a downloadable set of processor-executable instructions for performing the method of claim 47.

9. EVIDENCE APPENDIX

This Appeal Brief is not accompanied by an evidence submission under §§ 1.130, 1.131, or 1.132.

10. RELATED PROCEEDINGS APPENDIX

Pursuant to Appellant's statement under Section 2, this Appeal Brief is not accompanied by any copies of decisions.